A picture containing drawing, food

Description automatically generated

**Evaluating Kernel Bypass Network Libraries on Commodity Hardware**

by  
**Simon Curtis**

School of Information Technology and Electrical Engineering,

University of Queensland.

Submitted for the degree of Bachelor of Engineering (Honours)

in the division of Software Engineering

22/06/2020

Simon Curtis

s.curtis@uqconnect.edu.au

22/06/2020

Prof Amin Abbosh

Acting Head of School

School of Information Technology and Electrical Engineering

The University of Queensland

St Lucia QLD 4072

Dear Professor Abbosh,

In accordance with the requirements of the Degree of Bachelor of Engineering (Honours) in the School of Information Technology and Electrical Engineering, I submit the following thesis entitled

“**Evaluating Kernel Bypass Network Libraries on Commodity Hardware**”.

The thesis was performed under the supervision of A/Prof Marius Portmann. I declare that the work submitted in the thesis is my own, except as acknowledged in the text and footnotes, and that it has not previously been submitted for a degree at the University of Queensland or any other institution.

Yours sincerely

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Simon Curtis

# **Acknowledgments**

Thank you to my supervisor, A/Prof Marius Portmann for providing insight and guidance throughout the last year. Thank you, Paul Koch, for the initial idea behind this thesis and for letting me borrow your network card. And thanks to Leila Irons for keeping me motivated throughout the last semester.

**Abstract**

Packet capture at network speeds of 10Gbits/s and higher is not viable using commodity hardware and the Linux kernel without packet loss. This is due to the high overhead of the Linux networking stack. Several software frameworks are available that bypass the Linux networking stack and deliver packets directly to userspace for processing. These frameworks can vastly improve performance. This report looks at three high-performance software frameworks (DPDK, Netmap, PF\_RING\_ZC) and compares them on their ability to receive incoming traffic and update entries in a NetFlow table. It describes a NetFlow table architecture that is agnostic to its underlying capture framework, and that enables low latency inserting and exporting. It was found that all 3 frameworks provide great improvements over the standard Linux Kernel, with DPDK being the best performer, followed by PF\_RING\_ZC and then Netmap.

**Contents**

[Acknowledgments iv](#_Toc43719785)

[Abstract v](#_Toc43719786)

[List of Figures x](#_Toc43719787)

[List of Tables xi](#_Toc43719788)

[Table of Acronyms xii](#_Toc43719789)

[1 Introduction 1](#_Toc43719790)

[1.1 Background 1](#_Toc43719791)

[1.2 Purpose 2](#_Toc43719792)

[1.3 Aims 2](#_Toc43719793)

[1.4 Coverage 2](#_Toc43719794)

[1.4.1 In Scope 2](#_Toc43719795)

[1.4.2 Out of Scope 3](#_Toc43719796)

[1.4.3 Relevance 3](#_Toc43719797)

[2 Theory 4](#_Toc43719798)

[2.1 The Linux Kernel Networking Sub-System 4](#_Toc43719799)

[2.2 High Performance Techniques 5](#_Toc43719800)

[2.2.1 Fully Userspace Processes 5](#_Toc43719801)

[2.2.2 Pre-allocating Packet Buffers 6](#_Toc43719802)

[2.2.3 Polling Vs Interrupts 6](#_Toc43719803)

[2.2.4 Direct Memory Access 7](#_Toc43719804)

[2.2.5 Batch Processing 8](#_Toc43719805)

[2.2.6 Multi-queue Support (RSS) 8](#_Toc43719806)

[2.3 Frameworks 8](#_Toc43719807)

[2.3.1 DPDK 8](#_Toc43719808)

[2.3.2 Netmap 11](#_Toc43719809)

[2.3.3 PF\_RING\_ZC 11](#_Toc43719810)

[2.3.4 Other Frameworks 12](#_Toc43719811)

[2.4 NetFlow 13](#_Toc43719812)

[3 Literature Review 15](#_Toc43719813)

[3.1 Gallenmüller et al 15](#_Toc43719814)

[3.2 García-Dorado et al 15](#_Toc43719815)

[3.3 Sandia 15](#_Toc43719816)

[3.4 Wang et al 16](#_Toc43719817)

[3.5 Review Conclusion 16](#_Toc43719818)

[4 Methodology & Experiment 17](#_Toc43719819)

[4.1 Overview 17](#_Toc43719820)

[4.2 Simulator Setup 17](#_Toc43719821)

[4.3 Testing the Frameworks 20](#_Toc43719822)

[4.4 NetFlow Table Design 23](#_Toc43719823)

[4.5 Table Exporter Functionality 25](#_Toc43719824)

[4.6 Flow Visual Tool 25](#_Toc43719825)

[4.7 Collecting Results 27](#_Toc43719826)

[5 Results 29](#_Toc43719827)

[5.1 Overview of Results 29](#_Toc43719828)

[5.2 Linux Kernel 30](#_Toc43719829)

[5.3 DPDK 31](#_Toc43719830)

[5.4 PF\_RING\_ZC 31](#_Toc43719831)

[5.5 PF\_RING\_FT 32](#_Toc43719832)

[5.6 Netmap 32](#_Toc43719833)

[5.7 Effect of Multiple Flows 34](#_Toc43719834)

[5.8 Statistical Variance 35](#_Toc43719835)

[5.9 Success of NetFlow table and exporter 36](#_Toc43719836)

[5.10 Further Discussion 38](#_Toc43719837)

[5.10.1 Contents of test traffic packets 38](#_Toc43719838)

[5.10.2 Effect on computer resources 38](#_Toc43719839)

[5.10.3 Easy of setup 38](#_Toc43719840)

[5.10.4 Ease of Use 39](#_Toc43719841)

[5.10.5 Portability 39](#_Toc43719842)

[5.10.6 Configurability 39](#_Toc43719843)

[5.10.7 Licence vs Free 39](#_Toc43719844)

[5.10.8 Recommendations 39](#_Toc43719845)

[6 Conclusions 41](#_Toc43719846)

[6.1 Future Work 41](#_Toc43719847)

[6.1.1 Improved Hardware 41](#_Toc43719848)

[6.1.2 Larger number of flows 41](#_Toc43719849)

[6.1.3 Improvements to application software 41](#_Toc43719850)

[6.1.4 Improved Library Usage 42](#_Toc43719851)

[6.1.5 Other software frameworks 42](#_Toc43719852)

[6.1.6 Improved NetFlow functionality 42](#_Toc43719853)

[6.1.7 Effect on computer resources 42](#_Toc43719854)

[6.2 Summary 42](#_Toc43719855)

[7 Appendices 44](#_Toc43719856)

[7.1 A: Raw Results 44](#_Toc43719857)

[7.2 B: Repository Links 54](#_Toc43719858)

[7.2.1 Repository Links for all the frameworks and testing tool. 54](#_Toc43719859)

[7.2.2 Repository Links for Application code 54](#_Toc43719860)

[7.3 C: Method Setup Notes 55](#_Toc43719861)

[7.3.1 Configuring pktgen parameters 55](#_Toc43719862)

[7.3.2 Calculating the required packet count 56](#_Toc43719863)

[7.3.3 Calculating the measured throughput. 57](#_Toc43719864)

[7.3.4 Configuring point-to-point connection between two interfaces 58](#_Toc43719865)

[7.3.5 Notes on running pktgen 59](#_Toc43719866)

[7.3.6 Notes on Running Netmap Application 60](#_Toc43719867)

[7.3.7 Some Notes on Running PF\_RING\_ZC 62](#_Toc43719868)

[8 Bibliography 63](#_Toc43719869)

# List of Figures

[Figure 1: Linux Networking Stack [18] 5](file:///C:\Users\Simon\Documents\thesis\thesis-report\Curtis,Simon_44318923__ENGG4802.docx#_Toc43719870)

[Figure 2: Core Components Architecture [16] 9](file:///C:\Users\Simon\Documents\thesis\thesis-report\Curtis,Simon_44318923__ENGG4802.docx#_Toc43719871)

[Figure 3: Offered load vs Measured throughput for 64-byte packets. 30](file:///C:\Users\Simon\Documents\thesis\thesis-report\Curtis,Simon_44318923__ENGG4802.docx#_Toc43719872)

[Figure 4: Linux Kernel Raw Socket Packet Loss (Percentage) 31](file:///C:\Users\Simon\Documents\thesis\thesis-report\Curtis,Simon_44318923__ENGG4802.docx#_Toc43719873)

[Figure 5: Netmap Packet Loss (Log (Actual)) 33](file:///C:\Users\Simon\Documents\thesis\thesis-report\Curtis,Simon_44318923__ENGG4802.docx#_Toc43719874)

[Figure 6: Range of results as a percentage of the average value for those results. 35](file:///C:\Users\Simon\Documents\thesis\thesis-report\Curtis,Simon_44318923__ENGG4802.docx#_Toc43719875)

[Figure 7: Range of results as a percentage of the average value for those results. Sorted lowest to highest. 36](file:///C:\Users\Simon\Documents\thesis\thesis-report\Curtis,Simon_44318923__ENGG4802.docx#_Toc43719876)

[Figure 8: Range of results from testing multiple flows, as a percentage of the average value. 36](file:///C:\Users\Simon\Documents\thesis\thesis-report\Curtis,Simon_44318923__ENGG4802.docx#_Toc43719877)

# List of Tables

[Table 1: Packet Loss (%) at 10Gbits/s 29](#_Toc43719878)

[Table 2: Percent Packet Loss Reading Multiple Flow from PCAP File 34](#_Toc43719879)

# Table of Acronyms

|  |  |
| --- | --- |
| Acronym | Definition |
| API | Application Programming Interface |
| BPF | Berkley Packet Filter |
| CPU | Central Processing Unit |
| CRC | Cyclic Redundancy Check |
| DMA | Direct Memory Access |
| DPDK | Data Plane Development Kit |
| DTS | DPDK Test Suite |
| EAL | Environment Abstraction Layer |
| FT | Flow Table |
| GPU | Graphics Processing Unit |
| HPET | High Precision Event Timer |
| INET | Internet |
| I/O | Input / Output |
| IP | Internet Protocol |
| IPFW | IP Firewall |
| IRC | Internet Relay Chat |
| ISP | Internet Service Provider |
| NAT | Network Address Translation |
| NIC | Network Interface Card |
| NUMA | Non-Uniform Memory Access |
| OS | Operating System |
| PFQ | Per Flow Queuing |
| QOS | Quality of Service |
| RFC | Request for Comments |
| RSS | Receive Side Scaling |
| TCP | Transmission Control Protocol |
| UIO | User Mode Input Output |
| VLAN | Virtual Local Area Network |
| ZC | Zero Copy |

# Introduction

## Background

It is now commonplace for network speeds to reach 10-40Gbps or beyond in commercial environments such as businesses, government, hospitals, universities etc [1]. High-end network interface cards (NICS) can even reach speeds of 200Gbps [2]. These speeds are becoming so fast that it is hard for commodity hardware and software to keep up with network capacity, and they are constantly becoming bottlenecks in high-speed network operations.

Network operations such as routing, switching, network analysis, flow analysis, firewalls, intrusion detection systems etc, all require special hardware and equipment [3] to operate at these speeds without latency or packet loss. These solutions can be expensive and hard to set up, but the alternative of using commodity hardware and software like the Linux kernel often does not provide the required performance. This is because the kernel networking stack is slow, with multiple steps to pass packets up to userspace, copying data multiple times and wasteful context switching operations.

One technology that provides a workaround to this is kernel bypass. This, along with a variety of high-performance techniques can be employed to speed up the processing of packets by bypassing the kernel and allowing a userspace process to access packets directly from the NIC.

In recent years, several software frameworks have emerged that provide a variety of different services to improve networking speed on commodity hardware. Some of the more promising include DPDK, PF\_RIN\_ZC and Netmap. While these frameworks all differ slightly in implementation, purpose, features offered and availability, they all provide substantial speedup in packet processing.

This report develops a methodology to test and compare the packet processing ability of the frameworks mentioned above, in their ability to provide a capture framework for a partially working NetFlow exporter. It analyses the results of the measured throughput received by the NetFlow exporter, based on the underlying capture framework. The results analysed are dependent on several variables: offered load, packet size and number of flows.

## Purpose

The purpose of this report is to perform an investigation into three software frameworks (DPDK, PF\_RING\_ZC and Netmap) that provide high-performance network operations on commodity hardware. Secondly, this thesis will develop a basic NetFlow exporter implementation which is able to use these software libraries as its underlying capture framework. The software libraries will be tested on their ability to perform packet capture and insertion into the NetFlow table. They will be analysed based on their measured throughput into the NetFlow table. They will be tested with different traffic rates, packet sizes, and number of flows.

## Aims

1. To compare the packet capture performance of various available cutting-edge technologies at a high bandwidth on commodity hardware.
2. To test the limits of fast packet capture for commodity hardware running the Linux operating system.
3. To implement a NetFlow exporter which is agnostic to the underlying capture framework and easily interfaces to the different libraries.
4. To be able to populate a NetFlow table and export NetFlow at a line rate of 10Gbit/s.
5. To investigate the various techniques that are at the cutting edge of fast packet capture.

## Coverage

### In Scope

* Packet Capture
* Packet Processing
* NetFlow exporting
* DPDK, PF\_RING\_ZC, Netmap
* Kernel bypass and high-speed technologies
* Linux OS

### Out of Scope

* Packet forwarding
* Deep packet inspection / analysis
* NetFlow collection
* Creation of a low-level capture framework
* Using specialised hardware
* Other frameworks (Snabb, PacketShader, PFQ, BPF)
* Modifying the kernel stack

### Relevance

* 40Gbits/s networking exists today and is only getting faster. Software can't keep up.
* Performance hardware is expensive [3]
* Linux is a popular and practical OS in today’s computing world.
* These frameworks are the cutting edge of fast packet capture [4].

# Theory

## The Linux Kernel Networking Sub-System

The current networking stack for a standard Linux distribution can be split into 3 sections: the physical layer, user space and kernel space. The kernel space can be split into a further 5 sub-layers: System call Interface, Protocol Agnostic Interface, Network Protocol layer, Device Agnostic Interface and Device Drivers [5]. Each of these layers plays a role in moving a packet from the application process to the network device, and vice versa.

The system call interface allows a user process to access a socket via common read() and write() operations on a file descriptor. This is the starting point of the network stack.

The protocol-agnostic interface calls the sendmsg() function pointed to by the ops field of the socket struct. For example, in the case of the INET socket, it would call inet\_sendmesg().

The inet\_sendmesg() function will then call a protocol-specific function. For example, if the protocol used is TCP, then the kernel will now enter the TCP handling code, which breaks the packet up into chunks according to the TCP protocol and writes them to sk\_buff structures. The sk\_buff structures then have IP headers applied and are passed through various firewall checks and/or NAT masquerading.

The sk\_buff struct is now passed to the device-agnostic layer, where QOS (Quality of Service) and queueing routines are applied. The last step happens when the packet is dequeued and then the buffer is passed to the device driver when ready, for sending out the interface [6].

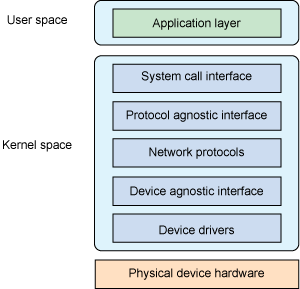


Figure 1: Linux Networking Stack [18]

The kernel, using this stack, provides a variety of services to applications wishing to utilise the network. Things such as checksum handling, routing, transport, network, link-layer protocols, interfacing to hardware and useful networking APIs such as sockets, are all offered to the application by the kernel. This make network programming very simple for application processes. However, using the default kernel networking stack has several limitations. For one, the main data structures used in the kernel for handling devices and packets (e.g. the sk\_buff structure which holds packet data) are constantly copied to different locations and structures [7]. This can be quite costly when done at high rates. Another downside is the fact that there needs to be a context switch from user mode to kernel mode. This wastes considerable CPU cycles when done frequently.

## High Performance Techniques

### Fully Userspace Processes

To provide some protection and safety, as well as provide some useful abstractions from lower-level details, an operating system will usually offer a set of system calls for application processes to communicate to underlying hardware (such as a network card) in a simple way. When a process wishes to use the underlying hardware, an application will call these systems calls, and the kernel will come in to do the work. However, when this happens, the CPU must suspend the currently running program, paging it into memory, along with all CPU registers. Then it will page in the kernel program from memory to perform the required task, i.e. reading something from the network card, and then reverse the process, putting previously running application back into the CPU registers and begin running it again. This process is called a context switch. It takes up valuable time that the processes could use to do other things such as running the program. If this sequence of event happens frequently, such as when a large amount of network traffic is hitting the NIC, then it could cause significant performance issues.

One way to get around this, is to have the application handle all the communication with network card in its own process space, removing the need for the kernel and eliminating context switches. This entails good performance benefits at the cost of more complexity.

### Pre-allocating Packet Buffers

Allocation memory is a costly application, as it involves a system call to the kernel and possible reshuffling of a processes memory layout [8]. In a high-performance operation, an application should avoid doing consistent memory allocation operations. This means it is better to allocation large portions of memory at the beginning of the process’s life, to hold enough space for all potential packets. In a normal application this would be undesirable as it would be an unnecessary waste of system resources, however in a high-performance operation, we can take advantage of excess memory to have packet buffers pre-allocated to save time.

### Polling Vs Interrupts

There are two main mechanisms that processes can use to check whether data is available for reading of writing on a device (for example checking if a packet has arrived on the network interface). These mechanisms are polling and interrupts [9]. Interrupts are a hardware device driven mechanism whereby the processor tells the device to notify it when it requires attention [10], e.g. a packet arriving on the network card. This involves some overhead as the interrupt has to initially go through the kernel, so there is a context switch, and also the processor must stop what it is doing and service the interrupt by running interrupt handling code. Polling is a protocol where a program periodically checks whether the device is ready to receive data or not [10]. When the frequency of an interrupt is low, it is more efficient to use an interrupt mechanism so that the CPU will not waste cycles periodically polling the device with no result. However, when the interrupt frequency is high, such as in this experiment where that packet arrival rate is in the millions of packets per second, the overhead generated by many interrupts ends up being less efficient that polling. Therefore, it is better in high performance operations to use polling rather than interrupts. It is obvious to see that it is strictly better to use a polling mechanism when the poll operation returns positive for each loop iteration.

Since the experiment described in this report is a high-performance operation it would be more efficient for network libraries to poll for incoming packets, rather than have constant interrupts occurring.

### Direct Memory Access

Direct memory access (DMA) is a technique where an I/O controller (which in the case of this report is the network card) can obtain direct access to a CPU's main memory while another program is running [10]. This is done by allowing the I/O controller to take control of the memory bus by specifying a memory address and allowing a block of memory to be read or written without involving the CPU. There are performance benefits in doing this as the CPU can do other tasks such as running another program whilst the controller is accessing memory. Also, it can significantly reduce the number of CPU cycles necessary for the transfer. Some high-performance network cards are even able to directly access a CPU's cache, which further improves performance [11].

A critical component of DMA is the DMA controller, which oversees the transfer of data between the IO controller and memory. When the processor wishes to initiate a DMA operation, it first issues a command to the DMA controller which contains the operation to perform (read or write), the IO device involved, the memory address to start the operation at, and the number of words to operate on. The controller then transfers the data, one word at a time without involving the processor. When the operation is complete, an interrupt is sent from the DMA controller to the processor. Since the processor is only involved at the start and end of the procedure, it is free to other tasks while the memory operation is being performed [11].

High performance networking libraries can take advantage of this technique be transferring packet contents into memory via DMA, leaving the process free to do other tasks such as processing the packet. This means that packet processing can happen concurrently while packets are still being read from the NIC, reducing packet loss, while still only using a single thread/core.

There is even further improvement to be gained when the IO controller can integrate with the DMA controller [11].

### Batch Processing

Batch processing is the idea of processing multiple packets at once. If you imagine the processing of a packet as a sequence of steps, a normal sequence of events would involve processing the first packet through step one, then step two and three etc, until completion. The idea behind batch processing, is instead of taking a single packet, the application would take a vector of received packets, and pass them all through processing step one, then, after they are all through step one, pass them all through step two. For each processing step, you call it on each of the packets before moving onto the next step. The reason behind this, is that it keeps the instruction cache 'hot' with the instructions from the current step, so that each step performed on a packet is faster than if the packets were passed through one at a time [12].

### Multi-queue Support (RSS)

Receive Side Scaling (RSS) is when the receiving NIC has multiple packet receiving queues for incoming traffic [13]. Each queue can be serviced by its own CPU, which effectively balances incoming packets amongst many CPU’s, enabling greater throughput. RSS is normally hardware based, meaning the queues will be internal to the NIC. This is the fastest method as it allows load balancing to be done at a hardware level. However, it is also possible to do this in software [14]. The NIC can either send packets to the multiple queues evenly, or it can use a hash function of packet header information to prioritise certain types of traffic.

## Frameworks

### DPDK

DPDK (Data Plane Development Kit) consists of a collection of data plane and networking libraries that allow for accelerated packet processing on a variety of CPU architectures and network cards. It is an open-source software platform originally developed by Intel and managed by the Linux foundation. It has a high degree of configurability [15].

DPDK uses an EAL (environment abstraction layer) (hides environment specifics from applications and libraries) to create libraries for specific environments. A user may link with a library to create their own applications. Other feature libraries independent to EAL are also offered. (i.e. for processing). All resources must be allocated prior to calling data plan applications (run to completion model). It does not support scheduler (will take up 100% of logical core it is running on) and all devices are accessed by polling [16].

The pipeline model is possible by-passing packets or messages between cores using rings. This allows work to be performed in stages and efficient use of code on cores (batch processing). Support for multi-process and multi-thread execution types and core affinity is also offered.

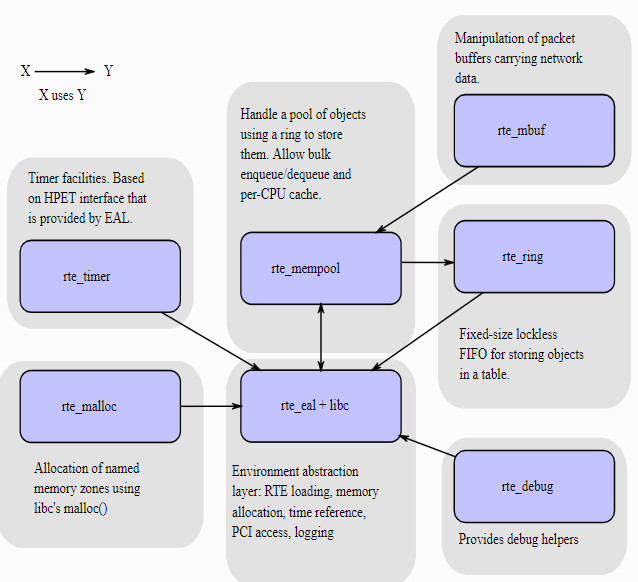


Figure 2: Core Components Architecture [16]

Core components of the DPDK architecture are [17]:

* rte\_timer:
  + Timer facilities based on HPET with the ability to execute functions and code asynchronously.
  + Enables periodic or single function calls.
* rte\_mempool:
  + Handling pool of objects in ring buffer.
  + Bulk enqueue/dequeue.
  + Per core object cache alignment.
* rte\_mbuf:
  + Manipulation of packet buffers.
  + Created at start-up time and stored in a mempool.
* rte\_ring:
  + When packet arrive on the network card, they are sent to a ring buffer.
  + Works on producer consumer model, where packets are placed onto one end of the buffer and the application checks regularly from the same buffer for new data. has separate pointer for producer and consumer.
  + Storing objects in a table.
  + Adapted to bulk operations.
  + Used for general communication.
* rte\_malloc:
  + Allocation of memory zones,
* rte\_eal:
  + Puts all the components together and allows communication.
  + Networking libraries.

DPDK uses a different driver to drive network cards (instead of ixgbe). It uses a UIO driver “uio\_pci\_generic” that allows the network controller to communicate directly to memory regions of a userspace program. It is a special kind of driver that aims to do most of its processing in userspace. Unlike Netmap, once this driver is loaded, the kernel is unable to access the network card until the new driver is unloaded again. This means that DPDK will take full control of the network card and other (non-DPDK) applications will not be able to use it until the driver is unbound [16].

DPDK must have hugepages enabled in the kernel to operate. This allows the network card to dump packet contents directly into the memory of the application process, effectively doing the same thing as DMA [16].

DPDK offers a variety of features on top of the basic networking operations. Some of these additional features include [18]:

* speed capabilities
* queues
* filters (VLAN filter)
* flow control
* flow API
* stats
* checksum offloading
* rate limiting
* traffic mirroring
* longest prefix matching

Some known applications that utilise DPDK as their underlying network stack are: DPDK Switch, and accelerated version of Open vSwitch, xDPd, a high-performance software switching solution, TRex, which is an open source traffic generator that uses DPDK, and DTS, which is a python based framework for functional tests and benchmarks.

### Netmap

http://info.iet.unipi.it/~luigi/netmap/ [19]

<http://info.iet.unipi.it/~luigi/papers/20120503-netmap-atc12.pdf> [20]

Netmap is a project that came out of the Universitá di Pisa, which was led by researcher Luigi Rizzo and supported by the European commission under a project called ‘CHANGE’. It is a framework for high speed packet I/O [19]. Netmap is implemented as a kernel module and collection of libraries that can be used as an API. It is available for Linux Windows and FreeBSD. Netmap still uses partial operating system network stack for low lever operations that are potentially dangerous such as driving the NIC and validating memory. However, it still makes use of well-known performance techniques: pre-allocating memory-mapping packet buffers, IO batching and its send and receive buffers are circular (ring) buffers (this matches the hardware implementation) [20].

Netmap supports most common network cards and will allow access to the network card. It supports libpcap for reading from and storing packet traces and says it can reach 10Gbit/s (line rate on a 10Gbit NIC). It is built in such a way that it is very easy to use and port applications that use raw sockets to Netmap because of its system-call like interface.

Some application that are built on top of Netmap are the vale software switch, click (a free software router) and IPFW (IP Firewall), for using a pc as a firewall.

### PF\_RING\_ZC

PF\_RING\_ZC is a kernel module built by Ntop for the Linux operating system [10]. It allows a high rate of packet processing and offers line-rate performance into the 10Gb/s [11]. The zero-copy capture interface does require a license, however free licences are available for educational research and non-for-profit purposes.

PF\_RING\_ZC does not use standard system calls, instead using its own functions and drivers. It offers a variety of features such as multicore support, and the driver can dump packet contents into memory where the application running on the CPU can access. As you can tell be the name, it is a ring buffer-based approach, similar to DPDK, and also offers a zero copy API, which improves performance by reducing unnecessary copying of data.

The zero-copy API is application and developer focussed, rather than hardware focussed and is easier to use than some of the other frameworks. Its network driver has a feature where you can switch between kernel bypass mode, and then back to the network stack when required. This makes it a very versatile technology with applications running on a personal computer easily switch between running high performance operation and having a full network stack.

Several applications exist on top of PF\_RING\_ZC, mostly other products produced by Ntop. For example: N2disk, a traffic recording and replay tool. Nprobe, which provides flow-based traffic analysis and Ndpi for deep packet inspection, among others.

### Other Frameworks

There are several alternative high-performance libraries available other than the ones analysed in this report. Snabb, PacketShader, PFQ, BPF, PF\_RING (vanilla), fd.io.

Snabb is a powerful packet processor, and has an easy to use design philosophy. It has good scripting support and is quite easy to get an application up and running. However, it is a fairly new technology and has not had as much adoption into applications.

https://dl.acm.org/doi/pdf/10.1145/1851182.1851207

PacketShader is a high-performance software router framework for general packet processing with a graphics processing unit (GPU). This report won't analyse PacketShader as it is more geared towards routing, 0which is out of the scope of this report.

N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, “On Multi–Gigabit Packet Capturing With Multi–Core Commodity Hardware,” in Passive and Active Measurement. Springer, 2012, pp. 64–73.

<https://link-springer-com.ezproxy.library.uq.edu.au/content/pdf/10.1007%2F978-3-642-28537-0.pdf>

PFQ is a new high-performance packet capture engine. It was designed with high parallelism in mind, as well as being connection and load balancing aware. Its design uses lockless queues, and it allows the application to define the level of parallelism, so that both legacy and natively parallel applications can benefit from it. It main advantage is that it can take complete advantage over a NIC that offers RSS. However, it is not a true kernel bypass technology.

<https://www.memsql.com/blog/bpf-linux-performance/>

The Berkley Packet Filter (BPF) is a technology that provides a raw interface that allows userspace programs access to the data link layer. It was excluded from this report as it is not a true kernel bypass technology, and its performance is not as good as the other frameworks that are being compared.

PF\_RING\_ZC also has a predecessor technology (PF\_RING vanilla) that is produced by the same company (Ntop) however since the zero copy version of the technology is strictly better than the vanilla version (for the purposes of this report), the vanilla version was not tested.

Fd.io is a vector packet processing library for high performance. It is worth looking into or anyone continuing this work, but due to time constraints, it did not get analysed in this report. The reason that DPDK, Netmap and PF\_RING\_ZC were chosen for analysis are because of their popularity, and also because they are the most competitive at the time of writing.

## NetFlow

https://tools.ietf.org/html/rfc3954

https://tools.ietf.org/html/rfc7011

NetFlow is a service that provides network administrators a tool to access information about flows that pass-through network elements (such as a switch or router). A flow is defined as a unidirectional sequence of packets that share a common set of properties. For example, downloading a file from a server could be seen as a single flow. In the case of this report, the packet metadata that defines a flow is the collection of five TCP/IP header fields: source IP, destination IP, source port, destination port and protocol (although in other implementations it may not be limited to this). For each flow a series of useful metrics are collected for analysis. For example, a running total count of packets and bytes in each flow, and the start and end time of each flow.

NetFlow provides a fine-grained look at traffic running through a network that enables highly detail and flexible analysis of a network and resource gathering method. Some of its main usages are in: network, application and user monitoring and profiling, security analysis, capacity planning, data warehousing, ISP billing, and data mining.

NetFlow works on an exporter/collector model. A NetFlow Exporter process will run on network devices and collect flow information from traffic passing through the device. The information is stored in a flow table and then sent to a collecting process in the form of a flow record once the flow has finished (determined by the TCP fin flag), or after a timeout has occurred. A collecting process receives flow records from one of more exporting processes and may exist on a different network device. The collecting process can then perform a necessary analysis or long term storage of the data. The communication process between a NetFlow exporter and collector is defined in RFC 3954 and is done via an application level protocol and Flow template records.

For this experiment a partial NetFlow exporter was implemented. It is capable of classifying incoming packets to different flows and storing them in a table with a running packet and byte count. However instead of exporting them to a collecting process, it just displays a real time visualisation of data passing through the interface.

# Literature Review

## Gallenmüller et al

https://www.net.in.tum.de/publications/papers/gallenmueller\_ancs2015.pdf

A comparison of the 3 frameworks (DPDK, PF\_RING\_ZC, Netmap), are looked at in a research paper from the Technische Universität München by Gallenmüller et al. It compares the frameworks on their ability to forward packets. The paper evaluates the frameworks both on measured throughput and latency. The results of this paper show that PF\_RING\_ZC and DPDK are significantly superior to Netmap in both throughput and latency, with DPDK slightly better that Pf\_ring. However, their report states that Netmap has the added advantage of "interface continuity and system robustness". The paper only compares the frameworks on their ability to forward packets, it does not look at their ability to do packet capture and analysis (which is what is required to implement a NetFlow table). The paper also does not look to compare the frameworks against the standard Linux kernel.

## García-Dorado et al

A research paper by García-Dorado et al, presents a comprehensive theoretical investigation of various capture frameworks (PacketShader, PFQ, Netmap & PF\_RING\_DNA). They test the effect of many different metrics on the measured throughput of the frameworks, such as number of CPU cores and packet sizes. The authors of this paper only analysed packet capturing capabilities and ignored other aspects of packet processing.

## Sandia

https://prod-ng.sandia.gov/techlib-noauth/access-control.cgi/2015/159378r.pdf

The research paper produced by Sandia National Laboratories looks at packet loss between Netmap, Pf\_ring and the Linux kernel when tested at high network speeds. The report also compares Netmap on two different operating systems: FreeBSD and Linux. The results show that the FreeBSD performs slightly better than Linux, but its results have much greater variance.

## Wang et al

<https://ieeexplore-ieee-org.ezproxy.library.uq.edu.au/stamp/stamp.jsp?tp=&arnumber=7883014>

This paper compares DPDK and PF\_RING on their traffic receiving ability only (without transmitting) and multithreading characteristics. They test specifically of Non-Uniform Memory Access Architecture (NUMA) platforms and focus on the run-to-completion (single thread) and pipeline (different thread for each processing step) models of packet processing. They analyse the effect on computer resources by introducing a variable ‘ρ’, the ratio of CPU time spent processing per unit packet. The paper for the run-to-completion model, DPDK is faster, and for the pipeline model, PF\_RING\_ZC is faster except for when inter-thread communication is used.

## Review Conclusion

The literature above shows that there is substantial knowledge about the performance of Linux networking and high-performance frameworks have been analysed thoroughly. However, most of these literary papers only compare the frameworks on their packet forwarding ability rather than in the case of this report where we are looking into the packet header fields and populating a NetFlow table. There are also only a few papers that compare multiple frameworks together, rather than exclusively.

# Methodology & Experiment

## Overview

The general format of the experiment was set up with 2 machines, one acting as a network simulator, which has the task of generating network traffic for the experiment, and the other was the test machine, which ran the experiments and recorded results.

This of course is not the only way that the experiment could have been performed. For instance, two 10Gbit network cards (or even a single one with two ports, such as the X540-T2) could be set up on the same machine, using one port for transmission and the other for receiving. Another idea could be to use virtual machines as the transmitting and receiving devices. However, both ideas introduce hidden variables that might not be taken into account when performing the experiment. For example, one of the systems that exists in computers and is part of the pipeline from network wire to CPU is the bus speed. Although this is not normally an issue, it still has a maximum bandwidth. For example, the Network card that is used in the experiment has a 8 lane PCI speed of 5.0GT/s. Although this is defiantly enough to handle 10Gbits/s, adding two ports to a single PCI slot is one extra thing that could become a potential bottleneck, so it is best to avoid things like this if possible.

Of course, another problem of using dual ports on a single machine or virtual machines is that the transmitting and receiving processes now have to content for CPU resources. All of these factors are things that we would want to minimise while testing. Having separate machines also closely mimics a real-world network, another reason why it was chosen as the test setup.

## Simulator Setup

One of the first problems encountered while designing the experiment was the problem of how to generate substantially fast network traffic. Since we did not have a real life 10Gbit/s+ network to test on, the author had to come up with a way to generate this amount of traffic with only the materials available to him. (Also, it probably wouldn't be wise to perform this experiment on a real network, as the results would be inconsistent. e.g. you wouldn’t have consistently sized packets, and the network might not always be running at one particular speed.). The original problem stated in this report is that a Linux kernel is not capable receiving 10Gbits/s traffic.

This is later shown in the results. So, from this, we can also safely presume that it is not able to generate high speed traffic either. Expensive and purpose build hardware might be possible in this situation, but due to budget constraints this was not possible in this case. Another solution might have been to generate traffic from multiple machines and forward and funnel to the test machine. Again, this would have needed more physical hardware, as well as a switch that could forward the traffic to the test machine. Also, this would have involved considerably more communication and co-operation and timing between the separate network simulators.

The option that remained was to use a single machine but use some software that was capable of doing what was required. Several packet generator tools were tested out, but there were very few that gave the required speed, were free, and had sufficient functionality. Eventually a packet generation tool "pktgen" was selected. This is a packet generation tool that is built on top of the DPDK framework. This raises the obvious question of: 'If the packet generation tool is built upon DPDK, then isn't that going to be a limiting factor during testing'. The answer is yes, the maximum bandwidth that we can test for is limited by the generation framework, which is also something we are testing. However, from the results we can see that this tool is defiantly capable of generating 10Gbit/s traffic, which was the goal of this experiment. Also, it is worth noting that the pktgen tool will only be generating packets and sending them off. The receiving software will have the added cost of implementing a working NetFlow table on top of this. This is the best solution that the author could come up with, but it should be noted for future work that the task of packet generation should be delegated to a tool that is not also being tested, if the budget allows.

DPDK's 'pktgen' tool is capable to generating 10Gbit wire rate traffic with 64-byte frames. It has a runtime environment to configure, start and stop flows, and can display real time metrics. To run pktgen, you first have to install DPDK on your simulator machine. Be aware that DPDK as certain bios, system and toolchain requirements for it too work properly. The critical specs for the pc that was used to simulate traffic are:

OS: Ubuntu Linux: 4.15.0-101-generic

CPU:

Memory:

NIC: X540-T1

For the full list of specs please see the Appendix. Note a critical requirement to run DPDK is to have the sse4\_2 CPU flag enabled. You also need at least 2 cores but preferable 4. A core i7 or better is recommended.

Once pktgen was running the method to perform the tests was:

1. set the size of the frames to be sent with the command:

set 0 size [size in bytes]

2. set the desired traffic rate with:

set 0 rate [rate in percent]

3. set the number of packets to be sent with:

set 0 count [number of packets]

3. when the test machine is ready to receive, start the transmission with:

start 0

4. once the current transmission rate drops to zero, (signifying the end of the test), stop

the port with:

stp

5. to verify the correct bytes and packets were sent, run:

page stats

The 0's in the above command signify configuring the 0th port. (The only port in this case). Below we will identify how the values for size, rate and count were calculated.

The size parameter is just iterated through the different packet sizes that were tested in this experiment i.e. iterated through {64, 128, 256, 512, 1024}. The rate parameter was iterated through the different rates that were tested (as a percent of 10Gbit/s) i.e. {1, 10, 50, 75, 100} or (100Mbit, 1Gbit, 5Gbit, 7.5Gbit and 10Gbit) respectively. This is because pktgen calculates the maximum rate it can send on the configured network card and takes the rate as a percentage of this. The count parameter is calculated from the rate and the size. Specifically, take the rate and calculate the number of bits that will be sent after transmitting for 10 seconds at that rate. (All tests were performed over 10 second periods). Then divide this by the 8 to get the number of bytes and divide again by the packet size to get the number of packets.

For running the multiple flow tests, pktgen was configured to read and replay from a pcap file. This pcap file was an excerpt of an IRC exchange, as well as some other background flows. To ensure that pktgen would run at the correct rate, it is critical to remove all the timestamps from the pcap file, otherwise pktgen will add these delays in. The same commands as before can be used to start the tests, so once pktgen is running and the test machine is ready to receive, start the tests with:

1. set 0 rate [rate in percent]

2. start 0.

At the same time as calling the start command. Start a timer for 10 seconds. Then run the stop command.

3. stp

Although this may seem like it introduces variance by manually starting and stopping the timer. It is not a major issue as you can still read the number of sent packets and bytes and use this to calculate what result you should be seeing on the test machine, and how much byte and packet loss has occurred. You can do this with

4. page stats

For example, after running page stats you may see that the number of bytes sent is \_\_\_\_\_ and the number of packets is \_\_\_\_\_. Comparing these to the results from the test machine for the number of bytes and packets received of \_\_\_ and \_\_\_, you can conclude that the packet and byte loss for that test is \_\_\_ and \_\_\_.

To make this process of running test simulations more efficient, you may want to utilise pktgen's scripting functionality via the Lua programming language.

## Testing the Frameworks

Below is described what happens on the test machine.

The process to describe what happens on the test machine can be split into 4 sections. (capture framework, base process, NetFlow table, exporter, visual tool). Most of the code written and used in this experiment is in C. There was a little bit of python used for the visualisation tool, and some shell scripting for automating testing and setup.

Firstly, there is the capture framework library. Included in here is the library functions, api, drivers and kernel modules for the capture frameworks that were used (DPDK, Netmap, PF\_RING\_ZC). For the default Linux kernel tests, this can be thought of as the kernel and system call interface. This is the code that is tasked with reading the packets off the NIC, either by polling the interface, or via interrupts / signals. This is an essential part of what we are testing in this experiment, i.e. how efficient and fast these frameworks are at network IO.

Although we do not directly modify the library framework code, we do use the systems and interfaces that they provide in our program. For example, we will call all initialisation functions necessary to set up the capture frameworks and will use their provided api for getting access to packet header and date pointers.

As an example, in each of the specific frameworks:

For the DPDK implementation we had to call: rte\_eal\_init, to set up the run time environment, rte\_eth\_dev\_count\_avail; to find the number of available port, rte\_pktmbuf\_pool\_create; to create the memory buffer pool, setup ports with rte\_eth\_rx\_queue\_setup, setup all thread needed, initialise the NetFlow table etc. The main loop with gets pointers into available packets looks like:

while (!force\_quit) {

for (i = 0; i < nr\_queues; i++) {

nb\_rx = rte\_eth\_rx\_burst(port\_id,

i, mbufs, 32);

if (nb\_rx) {

packet\_classify\_bulk (mbufs, nb\_rx, table);

for (j = 0; j < nb\_rx; j++) {

struct rte\_mbuf \*m = mbufs[j];

rte\_pktmbuf\_free(m);

}

pkt\_cnt += nb\_rx;

}

}

}

where 'packet\_classify\_bulk' is the function that does the packet process and eventually update the NetFlow table, although this section of the code will be discussed in a later section.

For Netmap the main library functions that are called include: nmport\_prepare, parse\_nmr\_config, nmport\_open\_desc. The main loop for receiving packets is:

1. for (i = targ->nmd->first\_rx\_ring; i <= targ->nmd->last\_rx\_ring; i++) {
2. int m;
3. rxring = NETMAP\_RXRING(nifp, i);
4. /\* compute free space in the ring \*/
5. m = rxring->head + rxring->num\_slots - rxring->tail;
6. if (m >= (int) rxring->num\_slots)
7. m -= rxring->num\_slots;
8. cur\_space += m;
9. if (nm\_ring\_empty(rxring))
10. continue;
12. m = receive\_packets(rxring, targ->g->burst, dump, netflow, targ->g->n\_table, &cur.bytes);
13. cur.pkts += m;
14. if (m > 0)
15. cur.events++;
16. }

where the recieve\_packets function takes a pointer to Netmap's ring buffer. This function eventually inserts an entry into the flow table.

The main program for the pf\_ring implementation looks similar. We start by calling the pfring\_zc\_create\_cluster library function. The device is opened with the pfring\_zc\_open\_device function and the license is checked with the fring\_zc\_check\_device\_license function. The main packet consumer thread gets a pointer to the next oacjet by calling pfring\_zc\_recv\_pkt\_burst. This pointer is then passed to the get\_netflow\_k\_v function to get a key value entry and then these are passed to the netflow\_table\_insert function to be inserted into the hash table (explained in more detail below).

The Linux kernel implementation work by using the raw socket interface that the kernel provides. The main system calls that are needed to for this setup are the 'socket' system call, the ioctl system call to set the interface to promiscuous mode and the 'recvfrom' system call to get a pointer to the next packet. Again, this is then passed to the relevant NetFlow table functions.

As part of the clean-up that happens when the process receives the interrupt signal telling it to close, it prints the total packet, byte and flow counts of all the entries in the flow table. It does this by looping over each list in each entry of the hash table. The reason for this is explain in more detail in a later section, but this is how the throughput into the table and the percent packet loss is measured.

## NetFlow Table Design

The NetFlow table that was implemented for use with the different frameworks was designed with a couple of constraints in mind. Firstly, it needed to be fast to insert and update flow entries. This should be obvious as performance is the primary metric of this report. It also needs to make sure it is fast as to not affect the performance of the capture framework. If it takes too long to insert an entry, then the framework being tested might drop packets that it otherwise could have read. Another constraint was that it had to be as consistent as possible when used across the different frameworks. This is to ensure that the comparison between the frameworks is as fair as possible. However, there are some exceptions to this case. Ntop, the organisation that created the PF\_RING\_ZC library, also have a native flow table implementation using the PF\_RING\_ZC table. Since using and testing this table turned out to be not a lot of work, an analysis of this table was included in this report, but it should be noted that this result was not using the same flow table as the other frameworks. Another exception was the DPDK flow table implementation, which I was able to modify from an existing code base. The code in this section has a few extra things that were not included in the flow tables for the other frameworks. Most of the differences are negligible, but it is worth noting one change that might be significant. The DPDK flow table was able to utilise hardware hashing functions whereas the other table implementation used software. It is assumed that the hardware hashing functions were faster than software, but it was not investigated into how much difference this made. Most other differences between the design of the flow table were minor and posed no significance. A third requirement of the flow table was that is it safe for multiple threads. This is due to the 'exporter' thread that periodically flushes the table to do something with the data. It is not a good idea that this thread has access to the table whilst other threads are inserting or updating entries. Similarly, it is important that only one thread has the capability to update the table at one time.

To conform to these requirements the NetFlow table was implemented as a hash table. This has the the properties of constant inserting time (as long as the number of flows is not greater than half the size of the table). Note that in the case of the tests that were run in this experiment, it was always the case that inserting into the table took constant time, as the number of flows in the test cases were always less that the number of Flow table entries, which is 1024 by default. In a real-world scenario, there may be more active flows than table entries. In this case, flows would double up, with multiple flows mapping to a single table entry. This is expected and would not corrupt the data as the flows would be separated in a list structure, however, performance may begin to slow. However, if optimal performance is still required with a high number of flows, the user could easily modify the default start size of the table to be whatever they desire, as long as computer resources permit.

The hashing function that is used to map flows to their corresponding hash table entries is a crc32 checksum of the 5-tuple fields described by a NetFlow 'flow'. This checksum is provided by hardcoded hashing tables in software, and by hashing recursively passing each field of the flow through the checksum.

idx = crc32c\_1word (key->proto, idx);

idx = crc32c\_1word (key->ip\_src, idx);

idx = crc32c\_1word (key->ip\_dst, idx);

idx = crc32c\_1word (key->port\_src, idx);

idx = crc32c\_1word (key->port\_dst, idx);

idx = idx % table->n\_entries;

As you can see by generating the table index using each of the fields in the flow, we can get a pseudo-unique index for each flow. It is also worth noting how the "key" struct is populated. This is done in the 'get\_netflow\_k\_v' function that takes a pointer to the packet header, and then traverses,

Once we have the index it is a simple matter of navigating to the offset and traversing the list (which we expect to have a length of 1 for a low number of flows), then updating the byte and packet counters for that flow.

## Table Exporter Functionality

Normally a NetFlow exporter would send expired flow data to a third-party collection service. For our table implementation, it was decided that it would be better to test the implementation of a Flow table a more simply way, by simply periodically writing out the flow table as a csv file.

There were several reasons for this. Firstly, it was for simplicity reasons. Although the process did not act like an exact or complete NetFlow exporter, it could be adapted to easily by simply changing what the exporting thread does. So, the proof of concept is there. Another reason is that it makes un-necessary work to go and set up some flow collection service, as it would not add anything to the project and is not within the project's scope.

The process of going exporting the flow table to a csv file was quite simple. Since the flow table was designed to be thread safe (see section above), a exporting thread was set up to run with a period of 1 second. This thread simply looped through each list in the hash table, copying the flow information, along with the current packet and byte count to a temporary buffer, separated by commas. Once all the entries had been added to this buffer, the lock on the flow table is released and the buffer is written out to a file. It is important that a temporary buffer is used and there is only 1 write operation so that the exporting does not hold up the task of inserting flows. Since writing is an IO operation and quite time expensive (relative to other computer operations), if multiply write's are done and the inserting thread has to wait for these to finish then it has to potential to drop a significant amount of packets as the underlying frameworks packet buffers fill up. It is also worth noting that the exporting thread does not do any processing of the flow data, such as sorting or filtering, for the same reason that it would be a bad idea to do any un-necessary work that may hold up other critical threads. This sort of processing work is handled by the program that reads from the exported file, and is described in the next section.

## Flow Visual Tool

To add a visual and practical element to this experiment, a real time flow visualiser tool was implemented to test the code for the flow table in its ability to export data, to make sure what is was storing is correct, and to add a use case for this experiment. Although a normal NetFlow exporter will export flows via the network into some third-party collection software, our implementation took a simplified approach. Instead of creating a full blown NetFlow exporter, the NetFlow table simplify exported itself to a temporary csv file, once per second (see section above for more details). This was a flexible approach that allows any other process to access this data and use it as it wishes. For example, a process could read the file and then export the flows via the network to a third-party collector, or (as in our case) implement a live table with packet and byte counts for the 10 biggest flows.

The visual tool is written in python and simply opens the exported csv file, sorts the entries by the number of bytes and then displays a table to the top 10 biggest flows. Doing this in a loop every 1 seconds gives a live updating table.

Source IP Destination IP Source Port Destination Port Protocol Packets Bytes

192.168.1.2 212.204.214.114 2848 6667 tcp 7124 398460

212.204.214.114 192.168.1.2 6667 2848 tcp 5862 4852461

192.168.1.2 71.10.179.129 4026 14232 tcp 1939 111254

71.10.179.129 192.168.1.2 14232 4026 tcp 1939 160889

172.200.160.242 192.168.1.2 11352 4984 tcp 1849 153177

192.168.1.2 172.200.160.242 4984 11352 tcp 1849 105001

192.168.1.2 68.206.150.243 1312 57322 tcp 1280 79794

24.177.122.79 192.168.1.2 8022 3863 tcp 1213 81276

192.168.1.2 24.177.122.79 3863 8022 tcp 1213 75429

192.168.1.2 67.71.69.121 1092 12492 tcp 903 43903

## Collecting Results

The format for running the tests and collecting the results followed a methodical procedure. Firstly, the correct packet count, size and offered throughput were calculated for the packet generation tool (pktgen). This is explained in more detail in section xxx. The setup script for the particular framework being tested is run, and then if there are no problems, the start script is then run. This means the program is not collecting packets and populating the flow table. Then

the start command is given to pktgen. The network test is now in progress. Once pktgen shows that the bandwidth has dropped back down to 0, the stop and page stats commands are issued, and an interrupt signal is sent to the main testing program. The stats for the number of packets sent and the number of bytes sent can be read from the stats page of pktgen. The test program should have outputted a sum of the total number of bytes and a total number of packets in all of the flows stored in the table. This way of measuring ensures accuracy that the number of packets and bytes recorded at the test machine have made it all the way through the test procedure, rather than just being read from the interface, which would be incorrect.

One more calculation must be done before however, because pktgen and the test process may calculate the number of bytes differently. Specifically, if packet gen is given a packet size of x bytes, it will count that packet as (x-4) bytes. As the 64 bytes includes an option VLAN field sent in the ethernet header. Since this field is set to null in pktgen, it is not included, and the actual size of the frame is 60 bytes. On the other end of the test setup, the ethernet header actually get stripped of before it is read by the NetFlow table code, and the table calculates the packet size from the IP header field. To verify the integrity of the above statements, one only needs to set up an experiment where pktgen sends only 1 packet, and then read the corresponding sent and received stats. Another difference is in the native flow table for PF\_RING\_ZC, which actually include the preamble and CRC check into byte counts as well. To counter this inconvenience, the results table must contain 'expected bytes' field, in which takes as an input the number of bytes sent as read from the pktgen stats and output the expected number of bytes to compare against.

Then the difference between the expected bytes and packets and the bytes and packets received may be used to calculate the measured throughput and percent packet loss fairly.

# Results

## Overview of Results

Overall, it was found that all the frameworks performed relatively well (compared to the standard kernel) with handling high speed traffic. All three frameworks (DPDK, Netmap and PF\_RING) were able to handle 10Gbit/s traffic with varying degrees of success and mostly produced acceptable packet loss, except in the extreme test case scenarios. In terms of absolute performance, it was found that DPDK was the best, followed closely by the PF\_RING\_ZC framework and the native flow table that it also provided. Netmap was the third best in terms of performance, although all three frameworks were very close.

The summarised results for the most extreme test case (a bandwidth speed of 10Gbit/s) for testing with a single flow can be seen below in Table 1.

Table 1: Packet Loss (%) at 10Gbits/s

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Packet Size** | **Linux** | **Netmap** | **PF\_RING\_ZC** | **PF\_RING\_FT** | **DPDK** |
| 64 Byte | 90.84 | 23.20 | 4.57 | 0.31 | 0.00 |
| 128 Byte | 83.94 | 0.07 | 0.10 | 0.17 | 0.00 |
| 256 Byte | 70.33 | 0.00 | 0.00 | 0.05 | 0.00 |
| 512 Byte | 47.40 | 0.01 | 0.00 | 0.00 | 0.00 |
| 1024 Byte | 25.49 | 0.01 | 0.00 | 0.00 | 0.00 |

Below is a graphical representation of another comparison of the different frameworks. This graph shows the measured throughput for each of the frameworks with respect to the offered load. It shows it with respect to 64 Byte packets, which are the most inefficient type of packet.

Figure 3: Offered load vs Measured throughput for 64-byte packets.

## Linux Kernel

The results show that the standard kernel for Linux was the worst performer, and verifies the original problem posed in the introduction that the kernel becomes the bottleneck in high performance operation on commodity hardware. The standard kernel handled the offered load well for the 100Mbit/s speed. At the 1Gbit/s speed it has small amounts of packet loss, but still within the acceptable range. The amount of loss occurring starts to be affected by packet size. From 5 Gbit/s onwards, the amount of packet loss increases substantially to where it would be unacceptable. Packet size has a very big impact, as at this speed 1024-byte packets seem to be handled fine still, but 64-byte packets occur a loss of 80%. Once the offered load is 10Gbit/s, even the 1024-byte packets receive substantial loss. Figure 4 shows a graphical representation of the packet loss for the Linux kernel.

Figure 4: Linux Kernel Raw Socket Packet Loss (Percentage)

## DPDK

The best performance framework according to the testing method was found to be DPDK, which had 0 packet loss through all testing scenarios, except in the case of 64byte packets at 10Gbit/s, for which it had insignificant packet loss of 0.0013%, and well within the acceptable range. There was no graphical representation for the results of DPDK as only a single test showed any packet loss. DPDK had no issues handling the case of multiple flows, with all tests showing no packet loss for those cases as well.

## PF\_RING\_ZC

The next best performer was found to be the PF\_RING\_ZC framework, which had either no or insignificant packet loss throughout most of the test cases. A graphical representation for PF\_RING\_ZC was not created as it would not be very useful. Please see Table 1 and Table 2 for a look at specific results of appendix A for the raw results. PF\_RING\_ZC handled packet sizes of 256 Bytes and larger without dropping a single packet. It dropped an insignificant number of packets at speeds of 5 and 7.5Gbit/s. The only area it struggled in was at the most extreme test case of 10Gbit/s with 64 Byte packets. Here we see a packet loss of 4.57%. When testing with multiple flows, PF\_RING\_ZC had no packet loss, except at the 10Gbit/s rate for which is had a 0.057% loss.

## PF\_RING\_FT

The native flow table that came with the PF\_RING repository (PF\_RING\_FT) was found to have great performance also and outperformed the NetFlow table built for this report that utilised the zero-copy framework for the single flow tests. However, the table build for this report (PF\_RING\_ZC) was better for the multiple flow case. It should be noted also that this analysis is technically not in the original scope, it was only included as extra because it required minimal configuring once PF\_RING was installed. PF\_RING\_FT performed extremely well on all tests, having a maximum of 0.3% packet loss across all test situations. Like DPDK and PF\_RING\_ZC, a graphical representation would not add anything of interest, as the results were quite uniform.

## Netmap

Netmap also held up well, with insignificant packet loss for most of the tests. However, for 64-byte packet size at 10Gbit/s, it had a packet loss percentage of 23.2%, which would be un-acceptable in a high-performance operation.

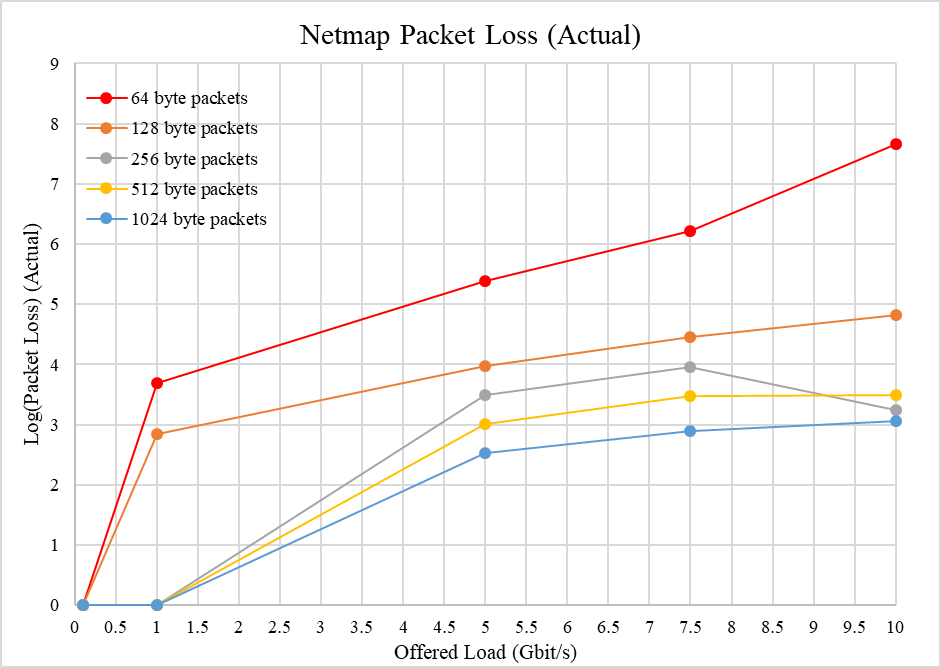


Figure 5: Netmap Packet Loss (Log (Actual))

Netmap also held up well in the multiple flow case, with a maximum packet loss of 0.056%.

## Effect of Multiple Flows

Table 2 summarises the results from testing with multiple flows, rather than a single flow.

Table 2: Percent Packet Loss Reading Multiple Flow from PCAP File

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Speed** | **Linux** | **PF\_RING\_FT** | **Netmap** | **PF\_RING\_ZC** | **DPDK** |
| 0.1 Gbit/s | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 1 Gbit/s | 1.18 | 0.00 | 0.00 | 0.00 | 0.00 |
| 5 Gbit/s | 67.75 | 0.00 | 0.05 | 0.00 | 0.00 |
| 7.5Gbit/s | 79.30 | 0.03 | 0.06 | 0.00 | 0.00 |
| 10 Gbit/s | 89.13 | 0.10 | 0.06 | 0.06 | 0.00 |

You can see that apart from when using the standard kernel is no significant packet loss here, and in fact there is even less than when testing with a single flow. This is because the packet trace that the generation tool was reading from for these tests was sampled from a real network. Because of this, the packets are not set sizes, they are random sizes. As seen in previous sections of results, the frameworks handle larger packet sizes much better than they handle smaller ones. From this, one can tell that the average packet size from the pcap file must be quite large.

The above table greatly show the effectiveness that using a kernel bypass framework would have in a real production environment, since these tests were generated from a real-life network capture.

One other interesting thing to note is the NetFlow table implementation for this report slightly outperforms the native flow table (PF\_RING\_FT). This contrasts with testing for a single flow. The reasoning behind this might be due to the internal representation of the different flow table. The one made for this report is done with a hash table, which allows constant access regardless of the number of flows. The implementation of the PF\_RING\_FT flow table is unknown, but it could be that the access time to it grows with the number of entries.

## Statistical Variance

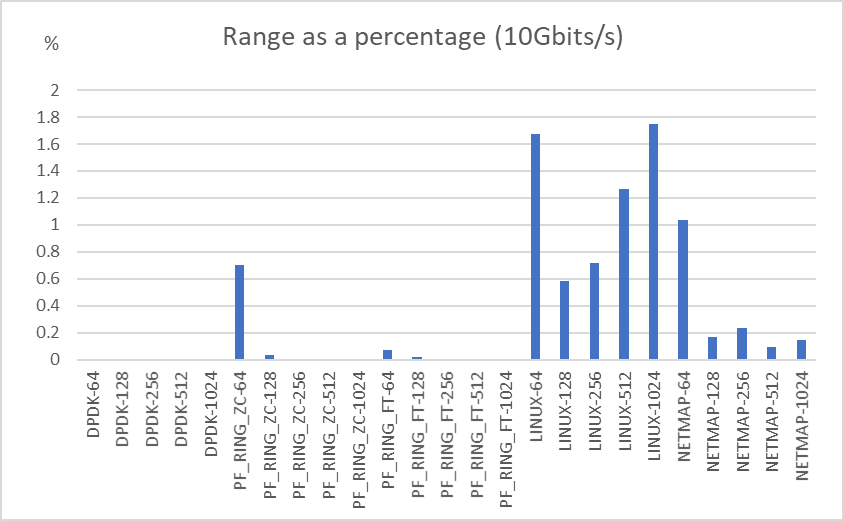


Figure 6: Range of results as a percentage of the average value for those results.

The variance between independent tests was analysed and appears to be quite low. The DPDK and PF\_RING\_ZC frameworks have almost non-existent variance in their results. This might be due to the fact that the results were upper bounded by the line rate of the network card. It appears that the variance is highest amongst tests with the 64-byte packet sizes. The Linux kernel had the most variance, and of the three other networks tested, Netmap had the most. It was not investigated into the reasoning behind why this variance was occurring. Figure 6 shows the range of results for each test as a percentage of the average value of the test, i.e. highest value – lowest value / average value. Figure 7 Shows the same thing but ordered lowest to highest for convenience. Figure 8 shows the same thing but for the multiple flow test case.

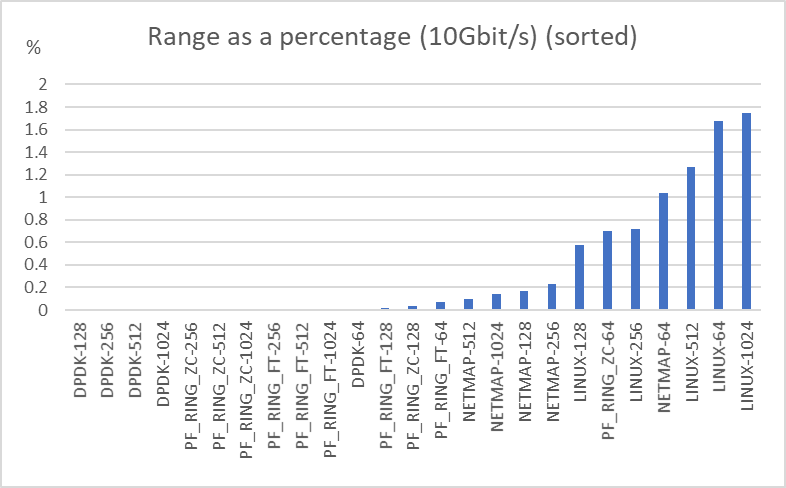


Figure 7: Range of results as a percentage of the average value for those results. Sorted lowest to highest.

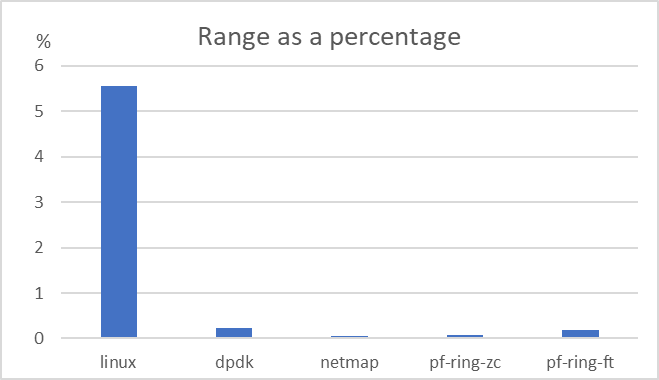


Figure 8: Range of results from testing multiple flows, as a percentage of the average value.

## Success of NetFlow table and exporter

The design of the capture agnostic NetFlow table was largely successful. It provided a feature that could use one of the underlying frameworks for a useful commercial or production application. The table was designed to have minimise the time taken to insert flow entries. This was successful because of the way that it was implemented as a hash table. There was minimal observed reduction in performance between initial testing of packet counting, to updating table flow entries from packet data.

The table was verified to be working correctly, as it reported a total number of packets exactly equal to the number reported by the traffic generation tool, minus the calculated packet loss. When testing with multiple flows, the total number of separate flow entries exactly matched that the number from the original pcap file used for generating traffic. Another verification was that the fields of the exported flow entries also matched to the fields of the pcap file. Finally, the fields and statistics recorded by the table matched up with those from the imported PF\_RING\_FT table. With all these factors together, it is reasonable to assume that the implemented NetFlow table was behaving functionally correct.

The NetFlow implementation is not a true one, for several reasons. Firstly, it doesn’t use the NetFlow protocol to export flows properly to a collector. It does not test incoming packets for TCP Fin flags and doesn’t have a timeout for each flow, only a global timeout. However, these issues would be able to be resolved easily and would be straightforward to do so. Implementing them would not add any extra noticeable delay to what is already there. The reason they were not implemented for this report are that is would be easier to test functionality without them, and the majority of tests would be dummy data from the packet generation tool anyway.

One place where the table *did* affect the performance was during the flow exporting period. The application code was written to export all the table entries once a second to a file on disk. Although this was written as a separate thread, to avoid interference with inserting entries into the table, there was still a slight bottleneck that was occurring. This was because a lock was placed around the flow table, to stop both the inserting and exporting thread accessing the table at the same time. This was critical, as the inserting thread could change some of the hash entries internal pointers, which could lead to a condition of the exporting thread crashing or giving undefined results. It was found that when exporting was turned on, a small performance drop was noticed. The exact amount seemed to depend on the framework, but it seemed to be around 0.1 – 2% more loss that without exporting. This was because while the table was being exported, no new entries could be added, and as the application could not service incoming packets, they were being forced to be dropped.

There are some ideas to improve this. The simplest, would be to reduce the frequency that the exporter thread runs. The current code does this once per second. But a one second resolution might not be needed for a lot of applications. Another idea could be to replace the global table lock with a lock specific to each entry. Then it will be less likely the two threads to require access to the same specific entry., and if they do, the lock will be released a lot quicker.

## Further Discussion

### Contents of test traffic packets

It may be worth talking about the contents of packets for the simulated traffic. For the single flow tests, the packet contents just contained filler data, it was just the characters a-z repeated for the entire contents. The protocol stack used in the test traffic was Ethernet -> IPv4 -> TCP. In this test case, the header fields for these were hard coded to specific values, and then the payload of the TCP was the filler content as described before. The reason for this was partly because it was the default of the test machine, and partly because the packet contents were not necessary to the tests and having a repeated packet with known contents was easier for debugging purposes.

In the case of the multiple flow testing, the packet contents were kept the same as in the pcap file.

### Effect on computer resources

The actual specific effect on the computer resources were not accounted during the testing process, however there was noticeable strain put on the system during the actual testing process (especially during the 10Gbit/s tests). This was different for each framework but was mainly seen as a huge spike in application memory allocated, and CPU utilisation jumping to 100% for several cores (the DPDK framework would have 3 dedicated cores to itself). It should be noted that a modern PC should be used to replicate these tests, with statistic consistent with or better than described in section xxx.

Another thing to note is the performance of the different frameworks for those that don’t require dedicated resources is somewhat heavily affected by other application currently running on the machine. This was noticed during testing when several web browser sessions were open on the machine that were not open during the previous round of testing. The performance was noticeably worse. Thus, it is important to make sure only minimal applications are active while testing is in progress.

### Easy of setup

Some of the frameworks were easier to install and setup than others when implementing the test setup. This could be an issue for a high-performance application depending on the requirements. DPDK was the framework with the most setup configuration required, with PF\_RING\_ZC and Netmap being a little easier.

### Ease of Use

If the interface that is being used by the network framework needs to be used for other things, this is easy to do with some of the frameworks than others. Netmap and PF\_RING do this the best, as to use them you need to load modified drivers into the kernel for the network card that you wish to use, but the kernel is still able to use these drivers, so once the application has finished, the NIC can be used as normal by the kernel again without any extra configuration. DPDK uses a special driver that cannot be used by anything else, so if you wish for another application to use the NIC, you must re-bind the NIC back to the original driver, which requires root privileges.

### Portability

Netmap is the most portable of the three software frameworks, as it compatible with the most hardware. It also took the least amount of effort to get working. DPDK took quite a lot of configuring to get working. It also has the most specific needs when it comes to hardware requirements. For example, it is designed for intel CPU’s, and must have a modern PC because it uses some modern instructions from the X86\_64 instruction set. PF\_RING\_ZC application can be portable, but you must obtain a license for each NIC you wish to use it on.

### Configurability

All the frameworks had many different parameters and configurations for setting up, function calling and application calling. This allows experienced users to take advantage of the software by using it exactly as required. Novice users will be ok with most of the configured defaults. DPDK has the highest degree of configurability among the three frameworks tested.

### Licence vs Free

It should be reminded that the PF\_RING\_ZC framework is obtained via licence from the company ‘Ntop’. This licence is free for all educational, research and non-for-profit institutions, but it comes at a cost for businesses. This is in opposition to DPDK and Netmap, which are all free and offer access to the full source code.

### Recommendations

For a high-performance network application looking to choose a capture framework ...

If absolute performance in a network operation is required, with cost, setup time, and usability not being an issue, DPDK is the recommended choice. It is the best performer out of the 3 tested and has the most configurability.

If cost is not an issue, but the application requires flexibility and usability in its setup and use, while still requiring high performance, then PF\_RING\_ZC is the recommended choice.

Netmap is a good choice if the application needs to be ported easily to many devices. It is also a easy setup choice for FreeBSD applications as it come pre-installed by default.

# Conclusions

## Future Work

Below are some ideas for improvements and advancements to the system and test results that somebody continuing from this report may like to explore.

### Improved Hardware

As you can see from the results the experiment did not test the limits of the frameworks and there is still room to improve. For example, DPDK can handle 10Gbit/s traffic easily, even at 64byte packet sizes. PF\_RING\_ZC also did not struggle throughout the tests. The limit that these frameworks can achieve has not been fully explored in this report. If one were to continue from this work, it is recommended they experiment with more powerful and capable hardware, such as a 25 or 40Gbit NIC.

### Larger number of flows

In this report, two different amounts of flows were tested. A single flow, and a replay of a packet trace with 180 flows. There was no notable performance reduction from testing with a single flow or multiple flows, and this was suggested to be because of the fact that the NetFlow table is implemented as a hash table, so it is expected that it would be constant access time if the number of entries is less than half the number of buckets in the table, which in the case of this implementation was 1024. One suggestion to enhance the testing in this regard is to do more tests in this area with a substantially higher number of flows i.e. in the thousands or tens of thousands. This will test the system in a new and interesting way.

### Improvements to application software

Improvements to the application software that runs the NetFlow table and sets up and calls the underlying library functions may be possible. For example, one situation that packet loss occurred in testing was during the periodic export process. This was because of a global lock that was placed on the entire NetFlow table, stopping another thread from inserting entries. Instead, it may be possible to use a separate lock for each entry, allowing the inserting thread to still run, and not cause holdups. Improvements like this may be able to increase the performance in testing.

### Improved Library Usage

Another possibility for future work that might further improve performance is to look at ways to utilise other features that the underlying software libraries offer. Most of the current code uses default fields for a lot of functions and does not explore some of the more advanced features and functions that the frameworks have to offer. This is especially true for DPDK, which has a large degree of configurability. For example, one could look at spreading the load across multiple cores, rather than just use a single core, which is what happens in the current code.

### Other software frameworks

Due to time constraints, this report was only able to analyse 3 different frameworks. Continuing from this work it would add to the completeness of the report to investigate some other competitive frameworks, particularly fd.io a vector packet processing framework, and Snabb.

### Improved NetFlow functionality

Another idea for enhancement is to improve the NetFlow exporting functionality. Currently, the table is not exported via the exact NetFlow protocol, and only exports the flows on a periodic timer, rather than looking for the completion of each flow, as a fully functional NetFlow export would. These features would add to the completeness of the project and allow for a product that may be useful in a production environment.

### Effect on computer resources

Another metric that could be worth gathering for analysis in future research is the effect that the different software libraries have on computer resources. E.g. the amount of CPU utilisation and allocated memory each one receives when it runs at maximum speed. This is important because resources may be less abundant on commodity hardware that if the application was running on a specialised machine. Also, the host machine might be doing a variety of other tasks, rather than just running the networking application.

## Summary

Networking speeds in todays commercial environments are quickly becoming faster than commodity hardware and software can handle.

This report aimed to look at a solution to the slow networking speeds that were offered by the standard Linux kernel. It looked at three separate high-performance networking frameworks, (DPDK, PF\_RING\_ZC, Netmap) that use a variety of advanced techniques, including kernel bypass. The goal of the report was to build a useful NetFlow exporter application, which can utilise these frameworks as its underlying packet capture tool. The NetFlow tool was built to be agnostic to the underlying framework, so that the performance of the different frameworks can be measured. The experiment aimed to achieve a rate of 10Gbit/s.

The experiment was designed to run on two physical machines, with one acting as a traffic generator and the other as a receiver. The traffic was generated for a DPDK tool, “pktgen”. The NetFlow table was designed as a hash table to allow fast update and insertion time.

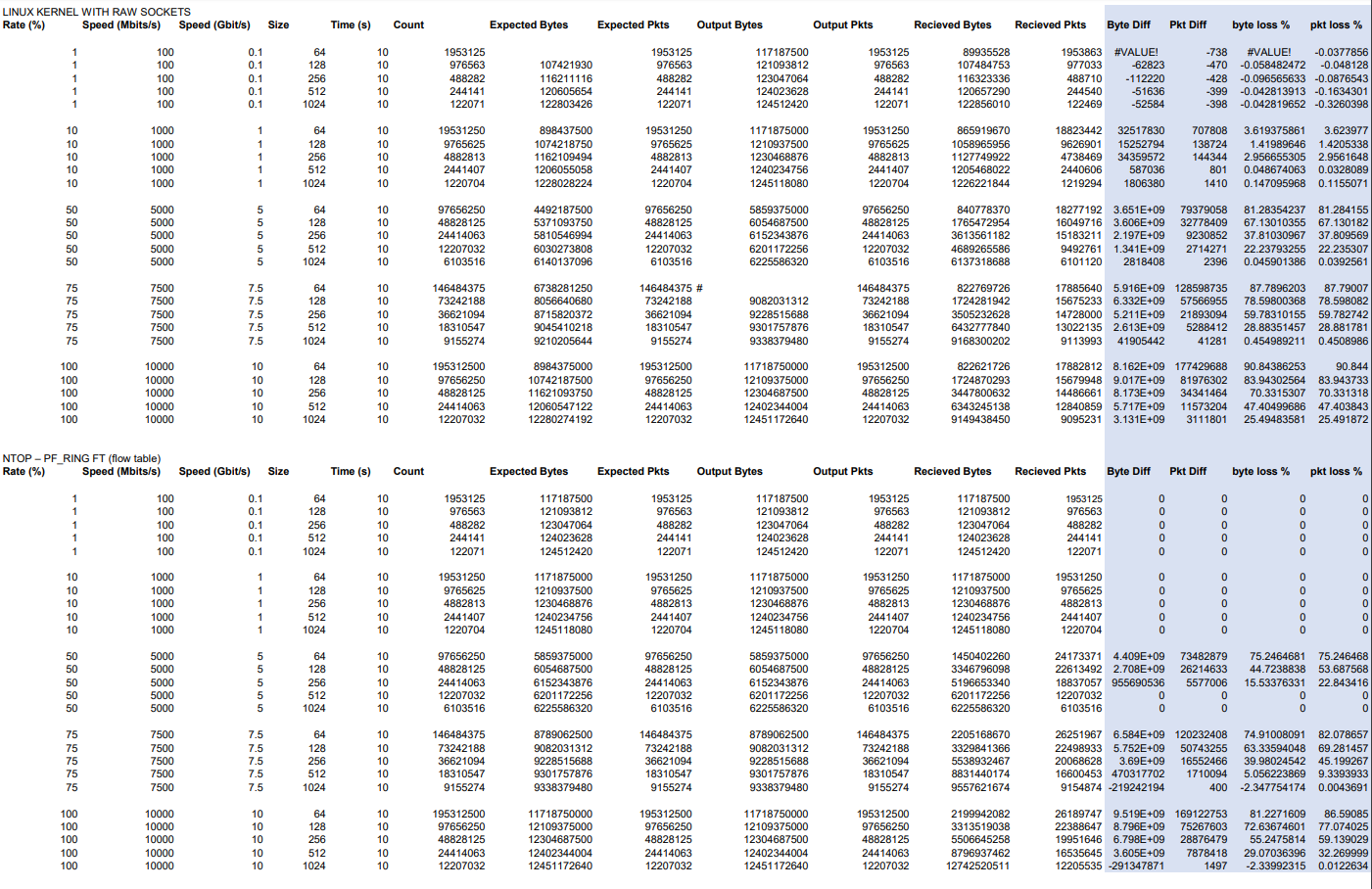
The results gathered from the experiment found that all three frameworks were vastly more capable to handle high speed traffic than the Linux kernel is. Packet size was shown to be a big influencer on performance, with smaller packets reducing performance significantly, especially at higher offered loads. The number of flows did not have much effect on performance. DPDK was found to be the best performing framework, followed by PF\_RING\_ZC and then Netmap.

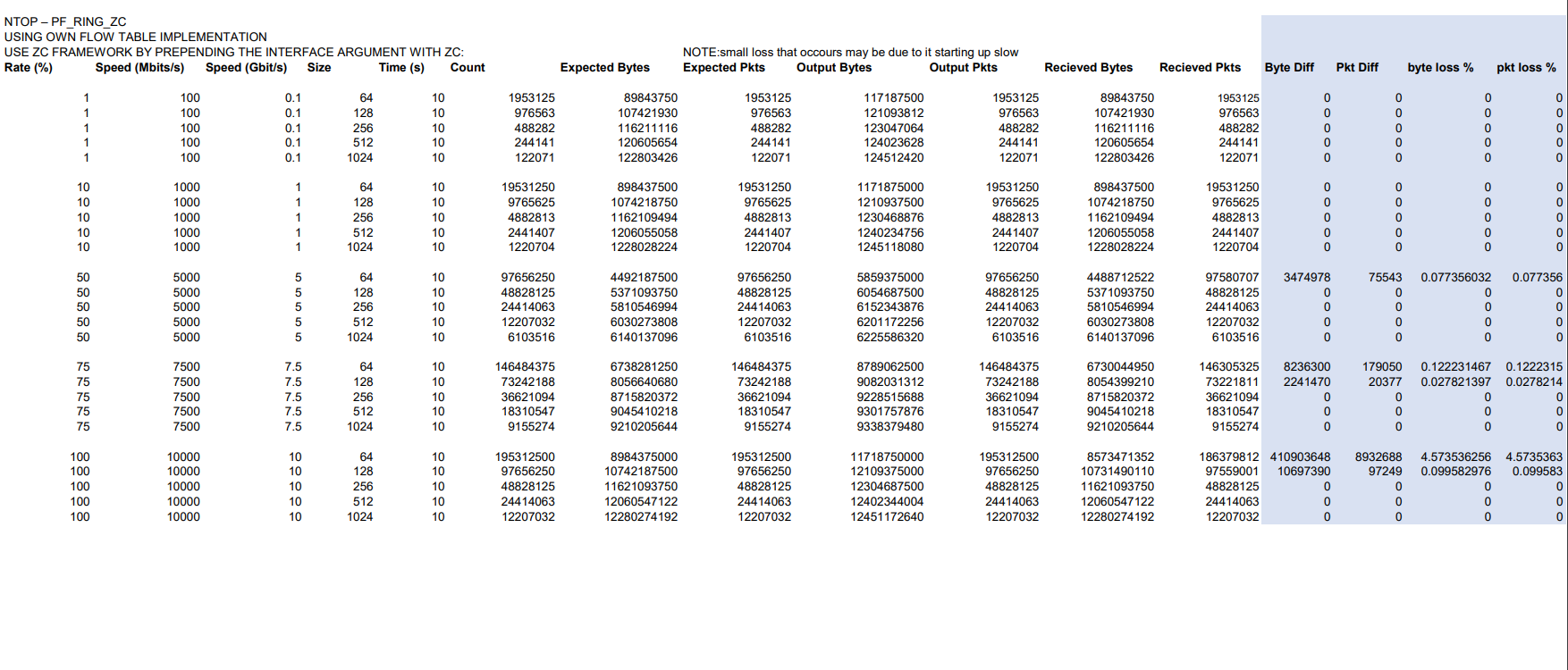
It was recommended that for applications where performance is critical, DPDK is the best choice. For applications that require a lot of flexibility and usability, PF\_RING\_ZC is a good choice. Netmap is the best choice for application that need easy setup and portability.

Some ideas to build upon the work presented in this report include enhancements to the NetFlow exporter, improvements to application and libraries, and enhanced and extended test cases.

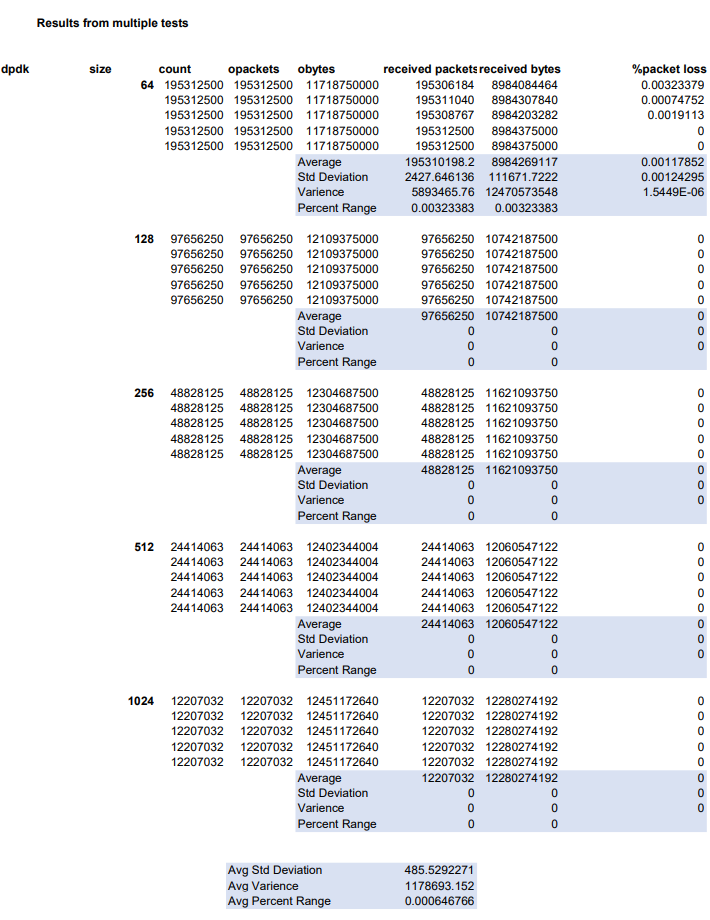
# Appendices

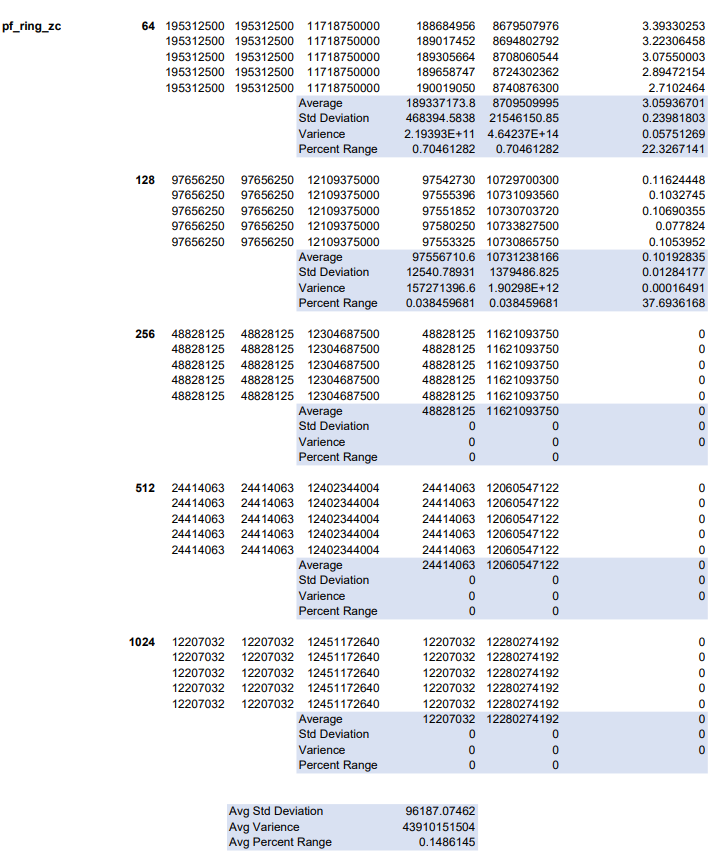
## A: Raw Results

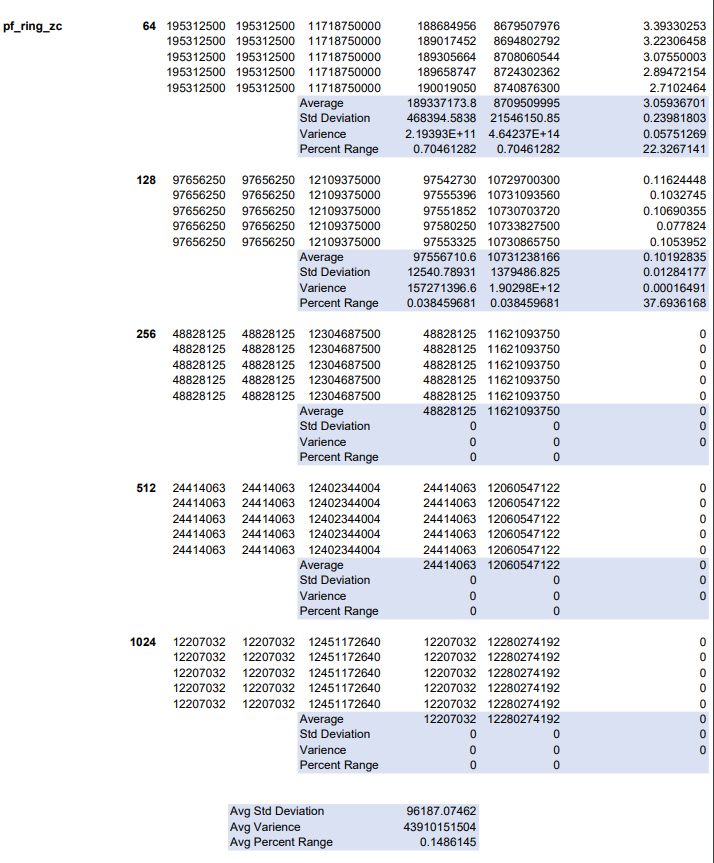


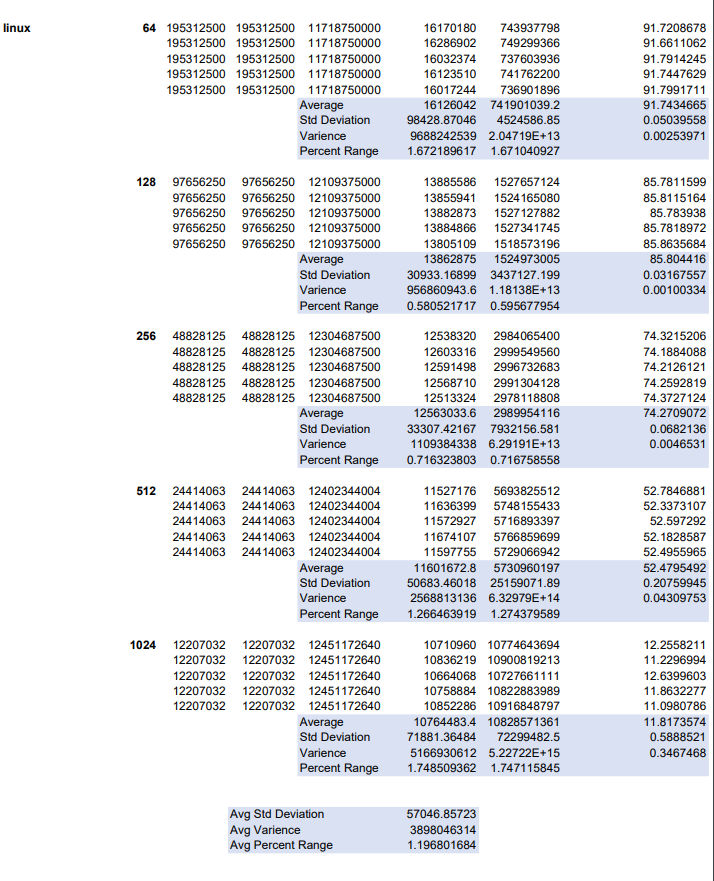


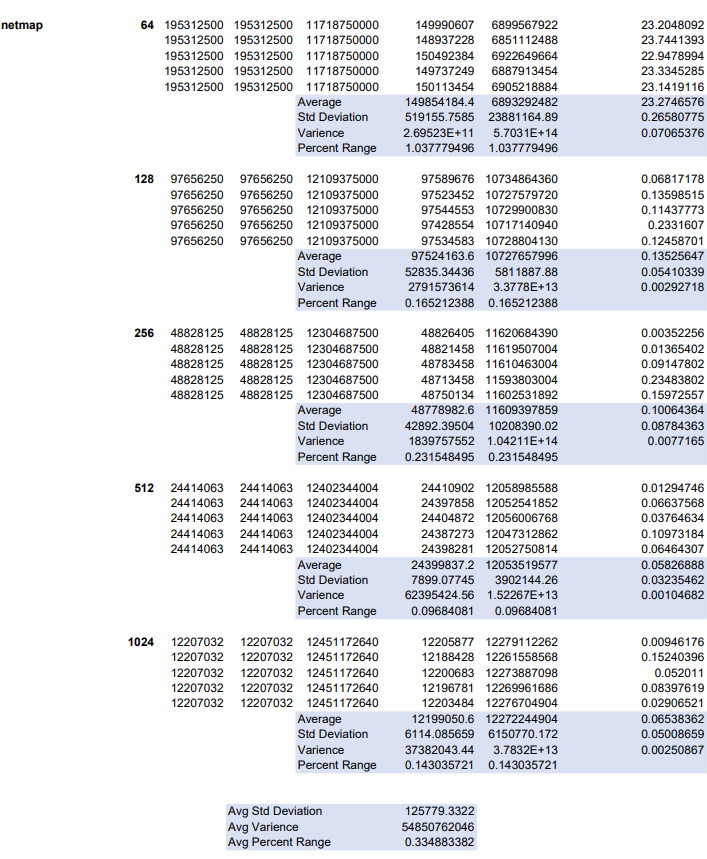


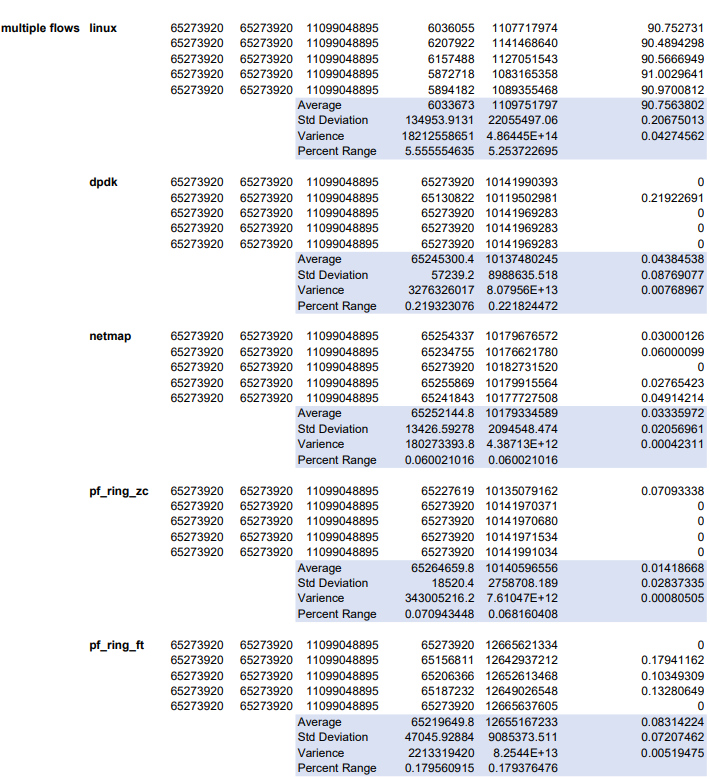












## B: Repository Links

### Repository Links for all the frameworks and testing tool.

DPDK libraries drivers and install: <https://github.com/DPDK/dpdk>

Packet generation tool “pktgen”: <https://github.com/Pktgen/Pktgen-DPDK/>

Libraries and drivers for PF\_RING\_ZC: <https://github.com/ntop/PF_RING>

Libraries and drivers for Netmap: <https://github.com/luigirizzo/netmap>

### Repository Links for Application code

NetFlow Application on top of Raw sockets: <https://github.com/scurtis142/raw-socket-netflow>

NetFlow Application on top of DPDK: <https://github.com/scurtis142/dpdk-netflow>

NetFlow Application on top of Netmap: <https://github.com/scurtis142/netmap-netflow>

NetFlow application on top of PF\_RING\_ZC: <https://github.com/scurtis142/pf-ring-netflow>

Application to visually display exported NetFlow data: <https://github.com/scurtis142/netflow-visual>

## C: Method Setup Notes

### Configuring pktgen parameters

A close up of text on a white background

Description automatically generated

### Calculating the required packet count

A close up of text on a white background

Description automatically generated

### Calculating the measured throughput.

A close up of text on a white background

Description automatically generated

### Configuring point-to-point connection between two interfaces

A close up of text on a white background

Description automatically generated

### Notes on running pktgen

A close up of text on a white background

Description automatically generated

### Notes on Running Netmap Application

A close up of text on a white background

Description automatically generated

A close up of text on a white background

Description automatically generated

### Some Notes on Running PF\_RING\_ZC

A close up of text on a white background

Description automatically generated

# Bibliography

|  |  |
| --- | --- |
| [1] | Intel, “Practical Considerations for Chossing a Network Speed for Business,” Intel, 2019. |
| [2] | P. Enberg, “On Kernel-Bypass Networking and Programmable Packet Processing,” Medium, 2019. [Online]. Available: https://medium.com/@penberg/on-kernel-bypass-networking-and-programmable-packet-pr ocessing-799609b06898. [Accessed 14 August 2019]. |
| [3] | Endace, “High speed packet capture solutions for Enterprises - Endace,” 2019. [Online]. Available: https://www.endace.com/endace-high-speed-packet-capture-solutions/enterprise/. [Accessed 12 August 2019]. |
| [4] | S. Gallenmüller, P. Emmerich, D. Wohlfart, D. Raumer and G. Carle, “Comparison of Frameworks for High-Performance Packet IO,” Technische Universität München, München, 2015. |
| [5] | J. Buse, “Linux Network Stack,” Linux.org, 2019. [Online]. Available: https://www.linux.org/threads/linux-network-stack.9065/. [Accessed 22 June 2019]. |
| [6] | S. Seth and M. Venkatesulu, TCP/IP architecture, design and implementation in Linux, Hoboken, N.J: Wiley, 2008. |
| [7] | R. Rosen, Linux Kernel Networking, New York: Apress, 2014. |
| [8] | D. Detlefs, A. Dosser and B. Zorn, “Memory Allocation Costs in Large C and C++ Programs,” Department of Computer Science, University of Colorado, Boulder, 1994. |
| [9] | TechDifferences, “Difference Between Interrupt and Polling in OS,” 3 January 2017. [Online]. Available: https://techdifferences.com/difference-between-interrupt-and-polling-in-os.html. [Accessed 6 May 2020]. |
| [10] | A. Butterfield and G. E. Ngondi, A dictionary of computer science, Oxford : Oxford University Press, 2016. |
| [11] | W. Stallings, Operating systems: Internals and design principles, Essex: Pearson Education Limited, 2018. |
| [12] | Fd.io, “The Vector Packet Processor (VPP),” Fd.io, 2018. [Online]. Available: https://fd.io/vppproject/vpptech/. [Accessed 10 May 2020]. |
| [13] | A. Choudhary, “Introduction to Receive Side Scaling (RSS),” Medium, 27 August 2019. [Online]. Available: https://medium.com/@anubhavchoudhary/introduction-to-receive-side-scaling-rss-7cd97307d220. [Accessed 8 May 2020]. |
| [14] | “DPDK Design Tips (Part 1 - RSS),” GalSagie, 26 Feb 2015. [Online]. Available: http://galsagie.github.io/2015/02/26/dpdk-tips-1/. [Accessed 1 May 2020]. |
| [15] | “About - DPDK,” DPDK, 2019. [Online]. Available: https://www.dpdk.org/about. [Accessed 13 Aug 2019]. |
| [16] | A. Yemelianov, “Introduction to DPDK: Architecture and Principles - Selectel Blog,” Selectel Blog , 2019. [Online]. Available: https://blog.selectel.com/introduction-dpdk-architecture-principles/. [Accessed 20 Aug 2019]. |
| [17] | Intel, “Part 1: Architecture Overview,” Intel, [Online]. Available: https://doc.dpdk.org/guides/prog\_guide/overview.html. [Accessed 5 April 2020]. |
| [18] | Intel, “2. Features Overview,” Intel, [Online]. Available: https://doc.dpdk.org/guides/nics/features.html. [Accessed 12 April 2020]. |
| [19] | T. Jones, “Anatomy of the Linux networking stack,” 2019. [Online]. Available: http://140.120.7.21/LinuxRef/Network/LinuxNetworkStack.html. [Accessed 22 August 2019]. |